编译大作业第二部分

小组成员： 唐溯珧 甘云冲 胡新宇 叶振涛

小组编号：13

问题描述

对于一个给定的表达式$Output = expr(Input_1, Input_2, \ldots, Input_n)$（$Output, Input_i$是张量或标量, $Expr()$表示用其参数构造一个表达式），我们如果已知了最终$loss$对于$Output$的导数$dOutput = \frac{\partial loss}{\partial Output}$，我们想知道$loss$对于某个输入的导函数是什么，也就是求$dInput_i = \frac{\partial loss}{\partial Input_i}$的问题。

通过对于输入表达式的编译分析过程，得到满足如下条件的求导表达式内容：

- 得到的求导表达式是一个或多个赋值语句形式
- 每个语句左侧的下标索引上不能有加减乘除等运算

自动求导技术设计

对于这样的问题可以拆分为以下几个步骤：

- 从Json格式的输入转换为树形表达式
- 通过IRMutate对语法分析树进行修改，对应的梯度表达式
- 解析输入生成符合规范的函数签名、梯度Tensor的初始化
- 进行表达式下标的替换，将左值的下标替换成不包含加减乘除的临时变量
- 打印对应的C代码

其中Project1已经可以做到从Json格式的输入当中解析表达式，构造语法树并将其打印为C代码。所以第一步和第五步都可以在Project1的基础上进行修改，第二步所涉及的梯度表达式的转换是这次Project的重点。

这里对于MoveStmt语句的右侧我们考虑采用如下的SDD规则进行修改，其中leftval即为左值，认为已知：

产生式	语义规则
$Expr \rightarrow Expr1 + Expr2$	Expr.grad = Expr("+", Expr1.grad, Expr2.grad); Expr.val = Expr("+", Expr1.val, Expr2.val);
$Expr \rightarrow Expr1 - Expr2$	Expr.grad = Expr("-", Expr1.grad, Expr2.grad); Expr.val = Expr("-", Expr1.val, Expr2.val);
$Expr \rightarrow Expr1 * Expr2$	Expr.grad = Expr("+", Expr("*", Expr1.val, Expr2.grad), Expr("*", Expr1.grad, Expr2.val)); Expr.val = Expr("*", Expr1.val, Expr2.val);
$Expr \rightarrow Expr1/Num$	Expr.grad = Expr("/", Expr1.grad , Num.val); Expr.val = Expr("/", Expr1.val, Num.val);

$Expr \rightarrow Var$	if(var.name == grad_name)
	{Expr.grad = Var("d"+leftval); Expr.val = Var.val;}
	else
	{Expr.grad = Expr(0); Expr.val = Var.val;}

$Expr \rightarrow Num$	Expr.grad = Expr(0); Expr.val = Num.val;

至此对于梯度推导的过程已经有了一个可行的实现方法，对于实现的具体细节在后续的**实现流程**一节当中进行描述。

样例说明

这里采用case5来进行说明正确性：

```
kernel: A<16, 32>[i, j] =  B<16, 32, 4>[i, k, l] * C<32, 32>[k, j] * D<4, 32>[l, j];
grad_to: "B"
```

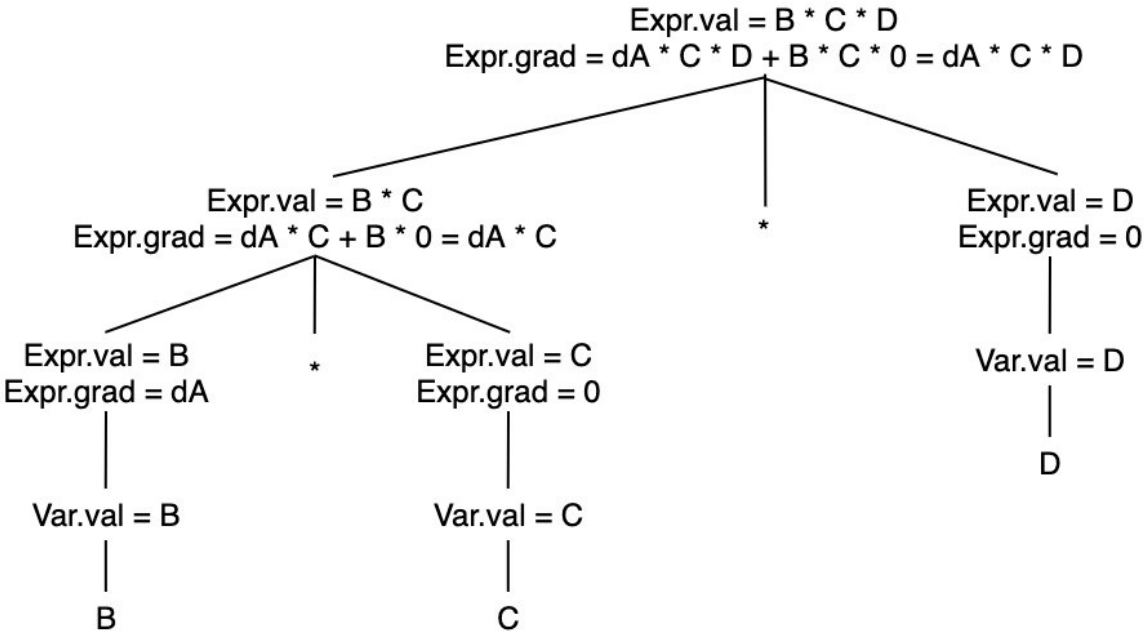
由于当前还不涉及下标的变换，所以在进行变换的过程当中，不会修改对应的下标，简单记作：

```
kernel: A = B * C * D
```

此时对应有：

```
leftval = "A"
grad_name = "B"
```

通过以上得到的SDD可以得到对应的注释语法分析树如下：



而直接通过链式法则求解梯度可以得到结果：

$$\frac{\partial Loss}{\partial B} = \frac{\partial Loss}{\partial A}\frac{\partial A}{\partial B} = dA * (B * C) = dA * B * C$$

可以发现所得到的结果是完全一样的，在这个样例当中正确性是可以保证的。

实现流程

初始化梯度tensor

在Kernel的最开始，对于需要`grad_to`的变量进行一个识别，然后通过调用`get_init`函数生成将其全部初始化为0的循环语句，加入kernel的语句列表当中。

IRMutate实现

在IRMutate的构造时传入两个参数：

```
    IRMutator(std::string val, std::string leftval) {
        grad_val = val;
        grad_leftval = leftval;
    }
```

其中val表示grad_to的对象，而leftval则代表MoveStmt的语句的左值。

每个visit返回的类型为`std::pair<bool, Expr>`，其中`bool`类型变量表示所得到的表达式是不是结果为0的表达式，便于之后清除冗余的内容，而Expr表达式得到的是对应的梯度表示，原本的表达式表示可以直接通过语法树节点获取。

```
    virtual std::pair<bool, Expr> visit(Ref<const IntImm>);
    virtual std::pair<bool, Expr> visit(Ref<const UIntImm>);
    virtual std::pair<bool, Expr> visit(Ref<const FloatImm>);
    virtual std::pair<bool, Expr> visit(Ref<const StringImm>);
    virtual std::pair<bool, Expr> visit(Ref<const Unary>);
    virtual std::pair<bool, Expr> visit(Ref<const Binary>);
    virtual std::pair<bool, Expr> visit(Ref<const Select>);
    virtual std::pair<bool, Expr> visit(Ref<const Compare>);
    virtual std::pair<bool, Expr> visit(Ref<const Call>);
    virtual std::pair<bool, Expr> visit(Ref<const Var>);
    ...
```

其中关键对于加减乘除进行梯度推导的实现在visit `Binary` 节点的过程当中进行：

```
pbe IRMutator::visit(Ref<const Binary> op) {
    if (op->op_type == BinaryOpType::Add) {
        pbe new_a = mutate(op->a);
        pbe new_b = mutate(op->b);
        if (new_a.first) return new_b;
        else if (new_b.first) return new_a;
        Expr t = Binary::make(op->type(), op->op_type, new_a.second, new_b.second);
        return mp(0, t);
    }
    else if (op->op_type == BinaryOpType::Sub) {
        pbe new_a = mutate(op->a);
        pbe new_b = mutate(op->b);
        if (new_b.first) return new_a;
        return mp(0, Binary::make(op->type(), op->op_type, new_a.second, new_b.second));
    }
  else if (op->op_type == BinaryOpType::Mul) {
    Expr a = op->a, b = op->b;
        pbe tmp_a = mutate(op->a);
        pbe tmp_b = mutate(op->b);
        if (tmp_a.first && tmp_b.first) return mp(1, Expr(0));
        if (tmp_b.first)
            return mp(0, Binary::make(op->type(), op->op_type, tmp_a.second, b));
        if (tmp_a.first)
            return mp(0, Binary::make(op->type(), op->op_type, a, tmp_b.second));
        Expr new_a = Binary::make(op->type(), op->op_type, a, tmp_b.second);
        Expr new_b = Binary::make(op->type(), op->op_type, tmp_a.second, b);
        return mp(0, Binary::make(op->type(), BinaryOpType::Add, new_a, new_b));
    }
  else {
        pbe new_a = mutate(op->a);
        if (new_a.first) return mp(1, Expr(0));
        return mp(0, Binary::make(op->type(), op->op_type, new_a.second, op->b));
    }
    return mp(0, Binary::make(op->type(), op->op_type, op->a, op->b));
}
```

在case10当中进行的是一个1D pooling的操作，三个输入变量的名称都是B但是有着不同的下标，所以需要被翻译为三个表达式。

```
{
    "name": "grad_case10",
    "ins": ["B"],
    "outs": ["A"],
    "data_type": "float",
    "kernel": "A<8, 8>[i, j] = (B<10, 8>[i, j] + B<10, 8>[i + 1, j] + B<10, 8>[i + 2, j]) /
3.0;",
    "grad_to": ["B"]
}
```

这里我们采用的方法是对于`grad_to`的变量，都增加一个后缀，当做不同的变量进行求导处理，所以以上的表达式就等同于：

```
    "kernel": "A<8, 8>[i, j] = (B_0<10, 8>[i, j] + B_1<10, 8>[i + 1, j] + B_2<10, 8>[i + 2,
j]) / 3.0;",
    "grad_to": ["B_0", "B_1", "B_2"]
```

对于三个变量分别调用IRMutate进行表达式转换，就可以得到对应的三个梯度表达式了，而对于多个同名同下标的变量。将其进行拆分求导再加和也不会改变最终结果，所以这种变换形式是可以保证正确性的。

函数签名生成

观察样例中要求的函数签名可以发现，函数签名的组织规律为

```
[inputs中求导后仍存在的变量，d[outputs]，d[grad_to]]
```

其中后两者可以直接根据读取到的JSON信息构造。对于第一部分，由于求导规律的特殊性，当目标求导变量在原kernel等式中仅为关于自身的一次项时，求导后的结果不会用到它本身，只会用到与它相乘的别的变量，即此时函数签名中不包含目标求导变量；特别的，如果出现了目标求导变量乘以目标求导变量的式子，则最后求导的式子中该变量也会保留，即函数签名中需要包含它。根据以上规律采用了较为简单的方法，第一部分的组成方式为：

{input in inputs if input != grad_to}∪{grad_to if it is nonleaner}

通过对kernel的简单字符串处理可以完成这一点，并把握好变量间顺序即可。如上明确了如何构造函数签名后，只需要沿用proj1中构造函数签名的函数，将其中一部分作为inputs，另一部分作为outputs，并修改printer使得多变量的输出能够被支持即可。

下标变换

针对每个语句左侧的下标索引中不能有加减乘除等运算的要求，我们对计算过程中所有包含加减乘除等运算的下标索引都进行下标变换，具体步骤为识别下标，命名替换，增加约束，去除冗余。

首先需要识别出包含加减乘除等运算的下标索引，我们利用Proj1中已有的设计，即函数parseVar中，会对输入计算式中的项进行拆分，下标索引的字符串表达式都存储在容器argnames中，对应的取值范围存储在容器shape中。这样一来我们只需要对argnames中的元素进行判定即可。

添加函数transform，输入为argnames中的表达式，利用已有的getitem函数获取其首个运算分量，若其不等于表达式本身，说明该下标索引表达式包含运算，识别成功。变量a存储我们新构造的临时变量的名字。

```
std::string a = "a_tmp";
bool transform(std::string str){
    std::string temp;
    int idx = 0;
    temp = getitem(str, idx);
    if (temp == str)
        return false;
    else{
        a[0] = a[0] + 1;
        return true;
    }
}
```

然后同样在函数parseVar中，如果发现表达式argnames[i]是包含运算的下标索引，便对其进行命名替换，更名为我们新构造的临时变量名a，然后添加等式约束，使该临时变量与原下标索引表达式等价，取值范围不变。

```
Compare::make(data_type, CompareOpType::EQ, parseArg(argnames[i], shape[i]), parseArg(a, shape[i]));
```

最后我们去除一些冗余的循环和约束，包括互相等价的临时变量以及Proj1中重复的条件判定。我们添加容器arg_tmp和arg_new分别记录每个循环过程中已经进行过变换的下标索引表达式和其等价的临时变量名，容器constraint记录已经生成过的条件判定语句，由变量名和其取值上界构成的pair<string, size_t>表示。每次进行下标变换或者添加条件判断前，先扫描容器，若已经存在等价的变换或者条件约束，直接取用即可，不然再进行下标变换/添加新的条件约束。

实验结果

对于case5得到的梯度求解方式如下（即之前用来说明的样例）：

```
#include "../run2.h"

void grad_case5(float (&C)[32][32], float (&D)[4][32], float (&dA)[16][32], float (&dB)[16][32][4]) {
  for (int ii = 0; ii < 16; ++ii){
    for (int jj = 0; jj < 32; ++jj){
      for (int kk = 0; kk < 4; ++kk){
        dB[ii][jj][kk] = 0;
      }
    }
  }
  for (int i = 0; i < 16; ++i){
    for (int j = 0; j < 32; ++j){
      for (int k = 0; k < 32; ++k){
        for (int l = 0; l < 4; ++l){
          if ((((i < 16 && j < 32) && k < 32) && l < 4)) {
            dB[i][k][l] = (dB[i][k][l] + ((dA[i][j] * C[k][j]) * D[l][j]));
          }
        }
      }
    }
  }
```

```
}
```

进行`./test2`的自动评分得到的结果如下：

```
➜  project2 git:(master) ./test2
Random distribution ready
Case 1 Success!
Case 2 Success!
Case 3 Success!
Case 4 Success!
Case 5 Success!
Case 6 Success!
Case 7 Success!
Case 8 Success!
Case 9 Success!
Case 10 Success!
Totally pass 10 out of 10 cases.
Score is 15.
```

可以看到十个case全部通过，得到了总分15分。

小组分工

唐溯珧：关于自动求导的IRMutate的编写，对于project1中语法树对应部分的调整

甘云冲：工作汇总编写实验报告

胡新宇：下标变换

叶振涛：函数签名的生成

实验思路设计及讨论由四人共同参与。

小组对Project1和Project2合并分工，两部分大作业在整体上尽可能保证分工均衡。